デバイスの記事

SystemCによるビヘイビア合成入門

第6回 テストベンチと階層構造

桜井 至

今回は，SystemCによるシミュレーション制御やテストベンチ，階層構造の記述方法について解説する．SystemCには多くのシミュレーション制御構文が用意されている．これらを利用しながらテストベンチを作成し，デバッグを行っていく．シミュレーションの実行にあたっては，Makefileを作成して，コンパイルなどの手間を減らす． （編集部）

前回（本誌2005年5月号，pp.114-120）と前々回（同2005年3月号， pp.123-130）は，ビヘイビア合成可能なC++記述について紹介しました．今回は，ビヘイビア合成とは直接関係はありませんが，ビヘイビア記述やビヘイビア合成後の検証に必要なSystemCによるシミュレーション制御，テストベンチ，階層構造に関する構文について紹介します．

SystemCにはクロックや階層構造の概念があるため，ハードウェア・システムの検証や設計に適しています．SystemCとビヘイビア合成を用いた設計フローを用いると，HDLの場合と比べて，より抽象度の高い設計を行うことができます．

● 多数のシミュレーション制御構文が用意されている

SystemCには，シミュレーションの開始や停止，シミュレーション時間の設定を行うための構文が用意されています．これらの構文を用いて，シミュレーションの実行を制御することができます．また，波形情報をVCD（value change dump）などのフォーマットでファイルに記録することもできます．

1）時間単位

シミュレーションの時間単位は，sc_time_unit列挙タイプとしてすでに定義されています．

```
enum sc_time_unit {SC_FS =0, SC_PS,
      SC_NS, SC_US, SC_MS, SC_SEC};
```

シミュレーションを実行する時間単位は，sc_set_default_time_unitで指定することができます．

```
sc_set_default_time_unit( 値 )
```

デフォルトの時間単位は1nsです．psで時間単位を指定したい場合は，以下のように記述します．

```
sc_set_default_time_unit(1, SC_PS);
```

sc_get_default_time_unit()によって，定義されている時間単位を知ることができます．また，sc_set_time_resolution(単位)で時間分解能を設定できます（デフォルトの分解能は1ps）．

2）sc_simulation_time

sc_simulation_timeでは，設定されたシミュレーション時間単位に基づいて，シミュレーション時刻を得ることができます．ビヘイビア合成では，スケジューリングの結果によって，アルゴリズムを処理するサイクル数（レイテンシ）が変わります．レイテンシと面積のトレードオフを考慮して適切なRTL（register transfer level）の設計を

リスト1　レイテンシ情報の取得

```
#define CLOCK_PERIOD 10
 . . .
double start_time, end_time;
start_time = sc_simulation_time();
 . . .
<シミュレーション・パターン入力>
<出力の観測>
 . . .
end_time = sc_simulation_time();
double latency = (end_time-start_time)/CLOCK_PERIOD;
```

選択する必要があるため，正確なレイテンシ情報の取得は重要です．

レイテンシについては，例えば**リスト1**のように`sc_simulation_time`を用いて，シミュレーション・パターンの入力から出力までの時間を測定して算出します．

3）sc_clock

`sc_clock`を用いてクロックを宣言します．クロックには，クロック周期，デューティ比，最初のエッジの遅れ時間などを以下のように指定します．また，実際にクロックを定義した例とその波形を**図1**に示します．

```
sc_clock 信号名(“名まえ”，周期，デューティ比，
                最初のエッジの時刻，エッジ・タイプ)
```

4）sc_time

`sc_time`を用いて時間変数の宣言を行うことができます．例えば，以下のように宣言します．

```
sc_time   cycle_time (10, SC_NS);
sc_time   finish_time (1, SC_US);
```

ここで，`cycle_time`は10ns，`finish_time`は1μsを表します．これらの時間変数は，`sc_clock`や`sc_start`などの関数で使用することができます．

5）sc_start，sc_stop

`sc_start`でシミュレーションを開始し，`sc_stop`でシミュレーションを終了します．

`sc_start`で時間を指定すると，指定した時間までシミュレーションを実行します．時間を指定しないと永久に実行するので，終了には`sc_stop`が必要です．また，`sc_start`に`-1`を指定すると，イベントがなくなるまで実行します．

```
sc_time finish_time(100, SC_NS);
sc_start(1, SC_US);
sc_start(finish_time);
sc_start(-1);
```

6）sc_trace

信号の波形情報をファイルに書き込んでおくと，波形表示ツールを使ってその波形を表示できます．波形のフォーマット（VCD，WIF，ISDBのいずれか）に従って，使用する関数を選択します．これらの関数は，指定したファイル名を示すファイル・ポインタを返します．

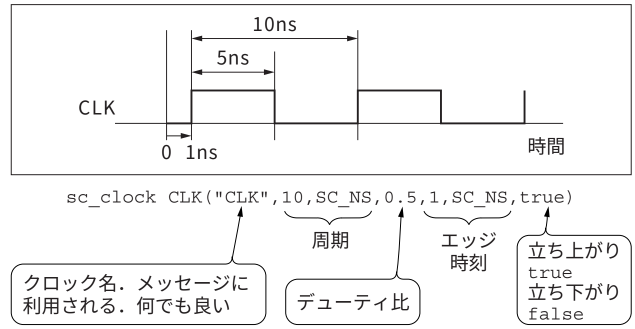


図1　sc_clockによるクロック定義
クロック波形は`sc_clock`を用いて定義する．何も指定しない場合は1ns周期で，デューティ比が0.5になる．

```
sc_create_vcd_trace_file
                 (“ファイル名”);  // VCD
sc_create_wif_trace_file
                 (“ファイル名”);  // WIF
sc_create_isdb_trace_file
                (“ファイル名”);  // ISDB
```

これにより，“ファイル名.フォーマット”という名まえのファイルが生成されます（例えばoutdat.vcd）．波形データを生成する信号または変数は，`sc_trace`関数を用いて以下のように指定します．

```
sc_trace(ファイル・ポインタ,
                  信号/変数,“表示名”);
```

シミュレーションを開始する前に，波形ファイルとトレースする信号を定義する必要があります．これらの関数は，`sc_main`内かコンストラクタ内に定義します．

```
sc_trace_file *tf = sc_create_vcd_
            trace_file(“sample”);
sc_trace(tf, clk, “clock”);
sc_trace(tf, reset, “reset sig”);
sc_trace(tf, data, “input data”);
```

● テストベンチを記述する

前回紹介したFIRフィルタを検証するテストベンチ例のヘッダ・ファイルを**リスト2**に，メイン記述を**リスト3**に示します．これらのテストベンチには，今回紹介したシミュレーション制御構文を使用しています．また，テストベ

リスト2　テストベンチのヘッダ・ファイル

```
#ifndef __TB_H_
#define __TB_H_

#include <systemc.h>

#define TAPS 5
#define TEST_PATS 32
#define CLOCK_PERIOD

static sc_uint<4> tb_coefs[TAPS] = {1,2,4,2,1};

SC_MODULE(tb) {

  // 入出力ポート宣言
  sc_in<bool>          clk;
  sc_out<bool>         reset;

  sc_in<bool>     input_ready;
  sc_in<bool>          output_valid;
  sc_in<sc_uint<8> >  output_data;

  sc_out<bool>      input_valid;
  sc_out<sc_uint<8> >  input_data;

  // メンバ変数の宣言
  sc_fifo<sc_uint<8> > expected;

  sc_uint<8> shift[TAPS];
  sc_uint<8> sum_of_coefs;

  // メンバ関数の宣言
  void stimulus();
  void checker();
  sc_uint<8> fir_filter_func(sc_uint<8> data);

  // ファイル変数の宣言
  ifstream fin;
  ofstream fout;

  // コンストラクタの定義
  SC_CTOR(tb)
  {
     SC_CTHREAD(stimulus, clk.pos());
     SC_CTHREAD(checker, clk.pos());
     fin.open(“input.dat”);
     if (fin.fail()) {
       cerr << “Can’t open file -- input.dat” << endl;
       sc_stop();
     }
     fout.open(“output.dat”);
     if (fout.fail()) {
       cerr << “Can’t open file -- output.dat” << endl;
       sc_stop();
     }
   }
};

#endif
```

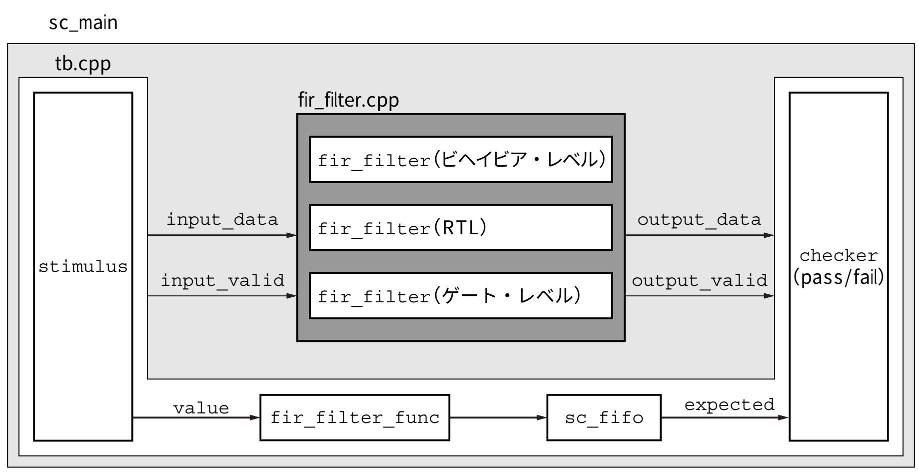


図2
テストベンチ記述例の構造
検証の対象となる設計は，前回紹介したFIRフィルタ．テストベンチでは，stimulusで入力パターンを生成し，checkerで期待値との比較を行う．期待値は，同じシミュレーション・パターンをビヘイビア記述に入力して生成する．検証対象となるモデルは，ビヘイビア・レベル，RTL，ゲート・レベルと，設計が進むに従って抽象度が変わる．SystemCを用いれば，一貫して同じテストベンチを用いることができる．ただし，ゲート・レベル表現を取り扱う場合は，HDLシミュレータとの協調シミュレーションを利用する必要がある．

ンチの構造を**図2**に示します．

このテストベンチは，二つのスレッドを持ちます．stimulusスレッドは，ファイルから入力パターンを順次読み込み，ビヘイビア合成の対象となる設計と期待値を生成する動作関数に，同じシミュレーション・パターンを入力します．関数からの期待値出力はsc_fifoで宣言したFIFO(first-in first-out)変数に蓄えます．FIFOを使用する理由は，ビヘイビア合成では入力から出力までのレイテンシが変化するので，具体的なクロック・サイクル数を指定できないためです．

検証対象となる設計(モデル)は，ビヘイビア・レベル記述，ビヘイビア合成後のRTL記述，もしくは論理合成後のゲート・レベル回路[注1]となります．SystemCを用いることで，異なる抽象度のモデルに対して，同一のテストベンチを使用することができます．

checkerスレッドは，出力値の観測と期待値との比較を行います．出力値は別のファイルに書き込みます．すべての結果が一致すれば，“Simulation Passed”というメッ

注1：SystemCではタイミングを含めたゲート・レベルのシミュレーションを行うことが難しい．そのため，Verilog HDLシミュレータとの協調シミュレーションを利用して，Verilog HDLのゲート・レベル・データを処理することが多い．

リスト3　テストベンチ・ファイル

```
#include “tb.h”

// 期待値を生成する関数（設計と同じ）
sc_uint<8> tb::fir_filter_func(sc_uint<8> data)
{
    sc_uint<16> tmp = 0;

    for (int i=TAPS-1; i > 0; i--) {
          shift[i] = shift[i-1];
    }
    shift[0] = data;

    for (int i=0; i<TAPS; i++) {
          tmp += shift[i] * tb_coefs[i];
    }
    return tmp/sum_of_coefs;
}

// シミュレーション・パターンを生成する関数
void tb::stimulus()
{
  sc_uint<8> value;
  sc_uint<8> result;

  sum_of_coefs = 0;
  for (short i=0; i < TAPS; i++) {
          sum_of_coefs += tb_coefs[i];
  }

  reset = 0;
  for ( int i = 0; i<5;  i++) {
       shift[i] = 0;
  }

  input_valid = 0;
  input_data = 0;
  wait();

  reset = 1;
  wait();

  // シミュレーション・パターンの入力
  for(int i=0; i<TEST_PATS; i++ ){

      wait_until(input_ready.delayed()==1);

      input_valid = 1;
    // ファイルからのデータ読み込み
      fin >> value;
      input_data.write(value);

    // 期待値データの生成
      result = fir_filter_func(value);
    // FIFOへの期待値データの書き込み
      expected.write(result);
  }
}

// 出力データをチェックする関数
void tb::checker()
{
  sc_uint<8> value;
  sc_uint<8> result;

  double start_time = 0;
  double end_time = 0;
  int total_errors = 0;

  wait_until(input_ready.delayed()==1);
 // 入力の開始時間
  start_time = sc_simulation_time();

  for(int i=0; i<TEST_PATS; i++) {
    wait_until(output_valid.delayed() == 1);

    value = output_data.read();
   // 出力データの表示とファイルへの書き出し
    cout << value << endl;
    fout << value << endl;
    // FIFOからの期待値データの取得
    result = expected.read();

    // 期待値との比較
    if ( value != result ) {
      cout << “CYCLE  “ << i << “: “;
      cout << “ Simulation Mismatch : output= “ << value;
      cout << “ expected= “ << result << endl;
      total_errors++;
    }
  }

  // レイテンシ（クロック数）の計算
  end_time = sc_simulation_time();
  double latency = (end_time - start_time)/CLOCK_PERIOD/TEST_PATS;
  cout << “Latency per data = “ << latency << endl;

  // 期待値との比較結果の表示
  if ( total_errors == 0) {
     cout << “¥n ***  Simulation Passed !! ***¥n”<< endl;
   } else {
     cout << “¥n ***  Simulation Failed !! ***”<< endl;
     cout << “        total error = “ << total_errors << endl;
  }
  // シミュレーションの停止
  sc_stop();
}
```

リスト4　C/C++のファイル操作

```
// iostreamによるファイル操作

#include <iostream.h>

 . . .

ifstream f_in;
ofstream f_out;

f_in.open(“input.dat”);
f_out.open(“output.dat”);

f_in  >> input.dat;
f_out << data_out << endl;


// FILEポインタを用いたファイル操作

#include <stdio.h>

 . . .

FILE*  fip;
FILE*  fop;

fip = fopen(“input.dat”,”r”);
fop = fopen(“output.dat”,”w”);

fscanf(fip, “%d”, &data_in);
fprintf(fop, “%d ¥n”, data_out);
```

セージを表示します．

テストベンチでは，ファイルからデータを読み込み，結果をファイルに書き込むことがよくあります．SystemCでは，結果の表示にcoutまたはprintfを使用します．ファイル操作に関する基本的な関数を**リスト4**に示します．

● 階層構造を記述する

SystemCでは，sc_mainが最上位の階層となります．sc_mainから，テストベンチと設計のインスタンスを呼び出し，信号(sc_signal)を用いて双方を接続した記述

リスト5　sc_main トップ・ファイル

```
#include <systemc.h>
#include “tb.h”
#include “fir_filter.h”

#define CLOCK_PERIOD 10
#define DUMP_ON 1

int sc_main(int argc, char *argv[])
{
  //クロックの定義
  sc_clock            clk(“clk”, CLOCK_PERIOD,
                               SC_NS, 0.5, 0);

  // 信号宣言
  sc_signal<bool>        reset;
  sc_signal<bool>        input_ready;
  sc_signal<bool>        input_valid;
  sc_signal<sc_uint<8> > input_data;
  sc_signal<bool>        output_valid;
  sc_signal<sc_uint<8> > output_data;

  // インスタンスの呼び出し
  fir_filter   fir_filter0(“fir_filter”);
  tb           tb0(“tb”);

  // インスタンス接続
  fir_filter0.clk(clk);
  fir_filter0.reset(reset);
  fir_filter0.input_ready(input_ready);
  fir_filter0.input_valid(input_valid);
  fir_filter0.input_data(input_data);
  fir_filter0.output_valid(output_valid);
  fir_filter0.output_data(output_data);

  tb0.clk(clk);
  tb0.reset(reset);
  tb0.input_ready(input_ready);
  tb0.input_valid(input_valid);
  tb0.input_data(input_data);
  tb0.output_valid(output_valid);
  tb0.output_data(output_data);

  // VCSファイルの生成
#if defined DUMP_ON
  sc_trace_file *vp
          = sc_create_vcd_trace_file(“fir_filter”);
  sc_trace(vp, clk, “clk”);
  sc_trace(vp, reset, “reset”);
  sc_trace(vp, input_ready, “input_ready”);
  sc_trace(vp, input_valid, “input_valid”);
  sc_trace(vp, input_data, “input_data”);
  sc_trace(vp, output_valid, “output_valid”);
  sc_trace(vp, output_data, “output_data”);
#endif

  // シミュレーションの開始
  sc_start();

  return 0;
};
```

リスト6　インスタンスの接続

```
SC_MODULE(alg) {
  sc_in<bool> i1;
  sc_in<bool> i2;
  sc_out<bool> o1;
   . . .
```

(**a**) SC_MODULEの例

```
alg
alg0(“alg0”);
alg0.i1(in1);
alg0.i2(in2);
alg0.o1(out);
```

(**b**) 名まえによる接続

```
alg alg0(in1, in2, out);
```

(**c**) 位置による接続

リスト7　SC_MODULEからの階層接続

SC_MODULEから別の階層モジュールを呼び出す場合，インスタンス呼び出しまたはポインタ呼び出しを用いる．インスタンスの接続はコンストラクタ内に定義する．名まえによる接続と位置による接続がある．

```
SC_MODULE(my_mod)
{
   sc_clock clk;
   . . .
   alg alg0;
   tb tb0;

   SC_CTOR :
     alg0(“alg0”),
     tb0(“tb0”),
     clk(“clock”,10,SC_NS)
   {
     alg0.clk(clk);
     alg0.i1(in1);
     alg0.i2(in2);
     alg0.o1(out);

     tb0.clk(clk);
     tb0.in1(in1);
     tb0.in2(in2);
   }
```

(**a**) インスタンス呼び出し

```
SC_MODULE(my_mod)
{
   sc_clock clk;
   . . .
   alg* alg0;
   tb* tb0;

   SC_CTOR :
     clk(“clock”,10,SC_NS)
   {
     alg0 = new alg(“alg0”);
     alg0->clk(clk);
     alg0->i1(in1);
     alg0->i2(in2);
     alg0->o1(out);

     tb0 = new tb(“tb0”);
     tb0->clk(clk);
     tb0->in1(in1);
     tb0->in2(in2);
   }
```

(**b**) ポインタ呼び出し

を**リスト5**に示します．また，この記述例ではsc_clockを用いたクロックの定義，sc_traceを用いた信号波形のダンプの定義，およびsc_startによるシミュレーションの開始の定義を示しています．

インスタンスの接続には，名まえによる接続と位置による接続があります．例えば，**リスト6(a)**のようなSC_MODULEに対して，名まえによる接続では**リスト6(b)**のようになります．また，位置による接続では**リスト6(c)**のようになります．

また，SC_MODULEから別の階層モジュールを呼び出すことができます．この場合，インスタンス呼び出しかポイ

リスト9
Makefile.defsの例

```
# SYSTEMC変数の設定（インストールしたツールの位置を指定）
SYSTEMC = /home/tools/systemc-2.0.1

INCDIR = -I. -I.. -I$(SYSTEMC)/include
LIBDIR = -L. -L.. -L$(SYSTEMC)/lib-$(TARGET_ARCH)
LIBS   =  -lsystemc -lm $(EXTRA_LIBS)
EXE    = $(MODULE).x

.SUFFIXES: .cc .cpp .o .x

$(EXE):         $(OBJS) $(SYSTEMC)/lib-$(TARGET_ARCH)/libsystemc.a
      $(CC) $(CFLAGS) $(INCDIR) $(LIBDIR) -o $@ $(OBJS) $(LIBS) 2>&1 | c++filt

.cpp.o:
      $(CC) $(CFLAGS) $(INCDIR) -c $<

.cc.o:
      $(CC) $(CFLAGS) $(INCDIR) -c $<

clean::
      rm -f $(OBJS) *~ $(EXE) core

ultraclean: clean
      rm -f Makefile.deps
```

ンタ呼び出しで別モジュールを呼び出し，接続はコンストラクタ内で行います．**リスト7**にSC_MODULEからの呼び出し例を示します．

● シミュレーション実行用のMakefileを作る

実際に，シミュレーションを実行するためには，OSCI（Open SystemC Initiative；SystemCの標準化・普及推進団体）のホームページ（http：//www.systemc.org/）からSystemCの実行に必要なクラス・ライブラリをダウンロードする必要があります．

SystemCファイルから，シミュレーションを実行する実行オブジェクトを生成するには，標準的なg++コンパイラを用います．そのため，市販のシミュレータを購入しなくてもSystemCファイルの検証環境を構築できます．

設計ファイルをコンパイルするために必要なMakefileの設定例を**リスト8**に，そこから読み込む共通的な環境を定義したMakefile.defsファイルを**リスト9**に示します．

ダウンロードしたデータには，各種のSystemC記述例が含まれています．これらを実行すれば，SystemCによる検証環境をより容易に理解できると思います．

リスト8　Makefileの設定例

```
# ターゲット・アーキテクチャ（linux，sparcOS5など）
TARGET_ARCH = linux

CC    = g++
OPT   = -O3
DEBUG = -g
OTHER = -Wall
EXTRA_CFLAGS = $(OPT) $(OTHER)
# EXTRA_CFLAGS = $(DEBUG) $(OTHER)

# 実行オブジェクトの指定
MODULE          = run
# コンパイルするターゲット・ファイルの指定
SRCS  = fir_filter.cpp tb.cpp main.cpp
OBJS  = $(SRCS:.cpp = .o)

#共通定義ファイルの指定
include  ../Makefile.defs
```

参考・引用*文献

(1) OSCI（Open SsystemC Initiative），SystemC User 's Guide / Functional Specification，2004.
(2) 長谷川裕恭，小川丈博，「連載・SystemCを使ったシステム設計」，Design Wave Magazine，2003年7月号〜2004年7月号.
(3) Thorsten Grotker，Stan Liao，Grant Martin，Stuart Swan（柿本勝，河原林政道，長谷川隆 監訳）；SystemCによるシステム設計，丸善，2003年.
(4) 大村正之，深山正幸；C/C++によるVLSI設計，共立出版，2003年.

さくらい・たかし
フォルテ・デザイン・システムズ（株）

<筆者プロフィール>
桜井 至．EDA業界に入り，HDLとしてはVerilog HDLから始まり，VHDL，Verilog-AMSへと進み，現在はSystemCを担当．さらに合成ツールとしては，論理合成，アナログ合成と進み，現在はビヘイビア合成を担当．設計言語と合成の大好き人間と言えるかもしれません．著書に「HDL設計入門」，「HDLによるディジタル設計の基礎」，「LSI設計の基礎技術」（テクノプレス社），および「Verilog-AMS入門」（CQ出版社）がある．現在，フォルテ・デザイン・システムズ（http：//www.forteds.com/）に所属．